Name: $\boxed{\text{Maura Kieft}}$
ID: $\boxed{103947905}$

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 7 **Spring 2019, CU-Boulder**

1. Multiple string alignment. As in class, we consider the operations of substitution (including the zero-cost substitute-a-letter-for-itself "no-op"), insertion, and deletion. An alignment of three strings x, y, z (of lengths n_x, n_y, n_z, respectively) is an array of the form:

$$\begin{array}{ccccccccccc} x_1 & x_2 & - & x_3 & - & - & x_4 & \ldots & x_{n_x} & - \\ - & y_1 & y_2 & - & - & y_3 & y_4 & \ldots & - & y_{n_y} \\ z_1 & - & - & z_2 & z_3 & - & - & \ldots & z_{n_z} & - \end{array}$$

That is, in the first row x appears in order, possibly with some gaps, in the second row y does, and in the third row z does. We define the cost of such an alignment as follows. The cost of a column

$$\begin{array}{c} x_2 \\ y_1 \\ - \end{array}$$

is the sum-of-pairs cost, e.g., in the preceding example we look at the cost of aligning x_2 with y_1 (a substitution, costing either 0 or 1 depending on whether $x_2 = y_1$ or not), the cost of aligning x_2 with $-$ (a deletion, costing 1), and the cost of aligning y_1 with $-$ (another deletion), for a total cost of $c_{subs} + 2c_{del} = 3$ for this one column of the alignment. The total cost of the alignment is the sum of the costs of its columns. Given three strings x, y, z, the goal of Multiple String Alignment is to find a minimum-cost alignment.

(a) (15 pts) Give an efficient (polynomial-time) algorithm for finding optimal alignments of three strings. Describe your algorithm in pseudocode and English, and prove an upper bound on its running time; you do not need to prove correctness (but you will only receive full credit if your algorithm is correct!). Hint 1: start with a recursive algorithm, where the recursion has 7 cases depending on what the last column of the alignment looks like:

$$\begin{pmatrix} x_{n_x} \\ y_{n_y} \\ z_{n_z} \end{pmatrix}, \begin{pmatrix} - \\ y_{n_y} \\ z_{n_z} \end{pmatrix}, \begin{pmatrix} x_{n_x} \\ - \\ z_{n_z} \end{pmatrix}, \begin{pmatrix} x_{n_x} \\ y_{n_y} \\ - \end{pmatrix}, \begin{pmatrix} - \\ - \\ z_{n_z} \end{pmatrix}, \begin{pmatrix} - \\ y_{n_y} \\ - \end{pmatrix}, \begin{pmatrix} x_{n_x} \\ - \\ - \end{pmatrix}$$

Hint 2: Your recursive algorithm will likely not be very efficient; what technique can you use from class to improve its efficiency?

help from geeksforgeeks.com
Depending on what the last column of the alignment looks like, we have seven cases.

Case 1: Cost(Xi, Yj, Zk) + A[i - 1, j - 1, k - 1]
Case 2: Cost(Xi, –, –) + A[i - 1, j , k]
Case 3: Cost(–, Yj, –) + A[i, j - 1, k]
Case 4: Cost(–, –, Zk) + A[i, j, k - 1]
Case 5: Cost(Xi, Yj, –) + A[i - 1, j - 1, k]
Case 6: Cost(–, Yj, Zk) + A[i, j - 1, k - 1]
Case 7: Cost(Xi, –, Zk) + A[i-1, j, k - 1]

Case 1, has three mismatches. Cases 2-4 have two mismatches and one insertion. Cases 5-7 have one mismatch and two insertions.
To find the cost for each column, we check the subcases:

· **Let C_{ij} = 0** If X(i) = Y(j) Else = 1

· **Let C_{ik} = 0** If X(i) = Z(k) Else = 1

· **Let C_{jk} = 0** If Y(j) = Z(k) Else = 1

Using these seven cases, we can find the optimal alignment of the three strings by recursively calculating the cost of each column until i, j, k = 0. Based off of this, we can calculate the sum-of-pairs cost for each column to find a minimum-cost alignment, by taking the min of the total cost of the alignment(i.e. the minimum of the sum of the cost of the columns) and checking if that cost is less than the previous column total (if less than, the column gets stored in low, if not, the minimum thus far stays in low) So, the minimum-cost alignment:

$$A[i,j,k] = min \begin{cases} A[i-1,j-1,k-1] + C_{ij} + C_{ik} + C_{jk} \\ A[i-1,j,k] + 2 \\ A[i,j-1,k] + 2 \\ A[i,j,k-1] + 2 \\ A[i-1,j-1,k] + C_{ij} + 2 \\ A[i,j-1,k-1] + C_{jk} + 2 \\ A[i-1,j,k-1] + C_{ik} + 2 \end{cases}$$

Initialization: A[0,0,0] = 0
n_x = i, n_y = j, and n_z = k

Pseudocode:

```
align(int i, j, k)
if((i! =0) || (j! =0) || (k! =0)) //for recursion until reach beginning of each string
    If (X(i) = Y(j))
        $C_{ij} = 0$
    Else
        $C_{ij} = 1$

    If (X(i) = Z(k))
        $C_{ik} = 0$
    Else
        $C_{ik} = 1$

    If(Y(j) = Z(k))
        $C_{jk} = 0$
    Else
        $C_{jk} = 1$

    A[i, j, k] = min(A[i-1, j-1, k-1] + $C_{ij} + C_{ik} + C_{jk}$,
                     A[i-1, j, k] + 2,
                    A[i, j-1, k] + 2,
                A[i, j, [k-1] + 2,
                A[i-1, j-1, k] + $C_{ij}$ + 2,
                A[i, j-1, k-1] + $C_{jk}$ + 2,
                A[i-1, j, k-1] + $C_{ik}$ + 2)
        alignment(i-1, j-1, k-1)
```

Run-Time:

For 3 strings X, Y, Z of lengths n_x, n_y, n_z respectively, we have n^3 cells in the cube, and for each cell top-left-front-corner we have to look at 7 different cases. If we carry out (n_x+1)x(n_y+1)-1 x (n_x+1)-2 = O($n_xn_yn_z$)= O(n^3) of them, we will expect an upper bound of O($7n^3$) Thus, the running time is O(n^3)

(b) (7 pts) Consider generalizing your approach to part (a) to k strings instead of 3. How many cases would there be to consider in the recursion? What runtime would the algorithm have? You do not need to give the algorithm, but must argue persuasively that your answers are correct, given your answer to part (a).

Generalizing my approach from part (a) to k strings of length n, will result in n^k cells in the cube. This will yield $2^k - 1$ cases to consider in the recursion, since we do not include the case containing no any value.

For example, in part (a), k = 3 so we had to consider $2^3 - 1 = 8$ - 1 = 7 cases. Which ended up with an upper bound of $(2^k - 1)(n^k) = 7n^3 =$ O(n^3).

Thus, for k strings, we will expect an upper bound of O($2^k n^k$)

2. The most efficient version of the preceding approach we know of for three strings takes an amount of time which becomes impractical as soon as the strings have somewhere between 10^4 and 10^5 characters (which are still actually fairly small sizes if one is considering aligning, e. .g, genomes). Because of this, people seek faster heuristic algorithms, and this is even more true for aligning more than three strings. One natural approach to faster multi-string alignment is to first consider optimal pairwise alignments and somehow use this information to help find the multi-way alignment. Here we show that naive versions of such a strategy are doomed to fail.

   Given an alignment of three strings, we may consider the 2-string alignments it induces: just consider two of the rows of the alignment at a time. If both of those rows have a gap in some column, we treat the pairwise alignment as though that column doesn't exist. For example, the alignment of x and y induced by the figure at the top of Q1 would be

$$\begin{array}{cccccccccc} x_1 & x_2 & - & x_3 & - & x_4 & \dots & x_{n_x} & - \\ - & y_1 & y_2 & - & y_3 & y_4 & \dots & - & y_{n_y} \end{array}$$

   Note that the column between x_3 and y_3 got deleted because it consisted of two gaps.

Name: Maura Kieft
ID: 103947905

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 7 **Spring 2019, CU-Boulder**

(a) (5pts) Give an example of three strings x, y, z such that the optimal alignment of x with y does not appear in *any* optimal alignment of x, y, z, and the same is true for optimal alignments of x, y and of y, z. Prove your example has this property. (An example where any optimal alignment of x, y, z does not contain an optimal alignment of x, y—but may contain optimal alignments of x, z and/or y, z—will earn you partial credit.)
(**question has been updated: answers below are for updated question**

x: A A A A T T T T
y: T T T T G G G G
z: G G G G A A A A

Theorem:
Given the 3 strings x, y, z, in any optimal alignment of x, y, z, at least one of the pairs (x, y) or (y, z) or (x, z) is not optimally aligned.

Proof:
Alignment of (x, y)

```
x: A A A A T T T T   =   x: A A A A T T T T * * *
y: T T T T G G G G       y: * * * T T T T G G G G
```

Alignment of (y, z)

```
y: T T T T G G G G       y: * * * G G G G A A A A
z: G G G G A A A A       z: T T T T G G G G * * *
```

Alignment of (x, z)

```
x: A A A A T T T T   =   x: * * * A A A A T T T T
z: G G G G A A A A       z: G G G G A A A A * * *
```

Therefore, we cannot combine the strings x, y, z into any consistent alignment and, thus, in this example using strings x, y, z at least one (if not all) of the pairs (x, y) or (y, z) or (x, z) is *not* optimally aligned.

(b) (7 pts) Prove that the gap between the cost of the pairwise optimal alignments and the pairwise alignments induced by an optimal 3-way alignment can be arbitrarily large. More symbolically, let d_{xy} denote the cost of an optimal alignment of x and y, and for an alignment α of x, y, z, let $d_{xy}(\alpha)$ denote the cost of the pairwise alignment of x and y which is induced by α. Your goal here is: for any $c > 0$, construct three strings x, y, z such that $d_{xy}(\alpha) > d_{xy} + c$ for any three-way alignment α. (Suppose all three strings have length n; how big can you make the gap c as a function of n, asymptotically?)

I don't know.

Name: Maura Kieft
ID: 103947905

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 7 **Spring 2019, CU-Boulder**

3. (14 pts) Give an efficient algorithm to compute the *number* of optimal solutions to the Knapsack problem. Recall the Knapsack problem has as its input a list $L = [(v_1, w_1), \ldots, (v_n, w_n)]$ and a threshold weight W, and the goal is to select a subset S of L maximizing $\sum_{i \in S} v_i$, subject to the constraint that $\sum_{i \in S} w_i \leq W$. For this problem, you will count the number of such optimal solutions. Your algorithm should run in $O(nW)$ time. If you only give an inefficient recursive algorithm, you can still receive up to 6 pts. (If you are having trouble solving this problem, you may want to start by developing an inefficient recursive algorithm and then seeing how to improve it using techniques from class.)

   Going off of the algorithm in class, the recursive formula allows for a memoization scheme to store and count the optimal solutions for the (k,w) subproblems. We initialize the algorithm with a table B with n rows and W+1 columns. We fill this table in one row at a time since the formula for B(k,w only refers to subproblems on the k-1 row. Using the three possibilities, we write a recursive expression which gives the optimal solution for k items and capacity w in order to find the number of the optimal solutions by denoting B(k,w) in terms of optimal solutions to smaller subproblems:

$$B(k,w) = \begin{cases} B(k-1,w) & if w_k > W \\ max(B(k-1,w), B(k-1, W-w_k) + v_k \end{cases}$$

   Pseudocode:

```
for w = 0 to W
      B[0,w] = 0
for k = 1 to n
      B[k,0] = 0
for k = 1 to n
     for w = 0 to W
           if(w[k] ≤ w
                 take = b[k] + B[k-1, w-w[k]]
                 leave = B[k-1, w]
                 B[k,w] = max (take, leave)
                 optimalSolutions++
           else
                 B[k,w] = B[k-1,w]
```

   After the algorithm recurses through the whole list, the variable optimalSolutions will return the number of optimal solutions to the knapsack problem. The correctness of this algorithm was proved in class, and it runs in O(nW) time.

4. Consider the following variant of string alignment: given two strings x, y, and a positive integer L, find all contiguous substrings of length at least L that are aligned (using no-ops = substituting a letter for itself) in some optimal alignment of x and y. Assume the costs of substitution, insertion, and deletion are given by constants $c_{subs}, c_{ins}, c_{del}$, and that the cost of substituting a letter for itself is zero.

   (a) (2 pts) Show that the output here contains at most $O((n-L)^3)$ substrings.

   Letting x and y be two strings, the algorithm will put those two strings into two loops in order to detect common strings and checks against the length of L in order to be less than and if the string is the same, it keeps adding it to a variable and if the string is not the same, add it to the result. The algorithm would normally run in O(n^3) time since we would use a while loop and a for loop for strings to calculate the sub-strings for the same values. But, since we have a condition that only allows length L sub-strings to be found, the loop will take n - L steps to calculate. Thus, time complexity is O($(n-L)^3$ in the worst case.

(b) (10 pts) Give an algorithm that solves this problem. You may use/modify/refer to the pseudocode for Knapsack from the lecture notes.

Letting x and y be two strings, we remove punctuation from both x and y and split the strings into words. Using two loops and a while looop to change the value of i in the loop, we detect the common strings. Then take the string to store values of the substring and check if is the same and keep adding it to the variable. If it is not the same, cheeck if there is already one contained in the variable and add that to the result.

Pseudocode:

```
i = 0
while ( i < x.length)          loops detect common strings // r = ” ” // if a common
string
d = i
if(r.length ≤ L)          for (j = 0; j < y.length; j++)                    if(x[i] == y[j])
//string same, keep adding to variable
            x+= 1
            i+= 1
      if x > 0 //check if there is already one and add to result
            z.append(r)
            i = d
            r = ” ”
            x = 0
if x > 0
      z.append(r)
      r = ” ”
      i =d
      x = 0
i+=1
```